CHORAL SOFTWARE · TECHNICAL INTERVIEW SERIES

ALGORITHMIC EDGE

A Pattern-Driven Path to Technical Interview Mastery

PYTHON 3 + JAVA



CHORAL SOFTWARE

Copyright

Algorithmic Edge: A Pattern-Driven Path to Technical Interview Mastery

Copyright © 2026 Choral Software

All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

This book is an original work. All problem scenarios, explanations, and source code in this book were independently written for this publication. Any resemblance to problems appearing on any other platform, book, or service is the result of both drawing on the same well-established, non-proprietary computer science concepts and techniques that are standard subject matter in the field — not of copying. Company names, systems, and scenarios used to frame problems are fictional and used solely for illustrative purposes.

Python® is a registered trademark of the Python Software Foundation. Java® is a registered trademark of Oracle and/or its affiliates. This book is not endorsed by, sponsored by, or affiliated with the Python Software Foundation, Oracle, or any other trademark holder mentioned herein. All trademarks are the property of their respective owners and are used in this book solely in a descriptive, nominative sense to identify the programming languages covered.

This publication is intended to provide accurate and authoritative information regarding technical interview preparation. It is sold with the understanding that the publisher is not engaged in rendering professional advice specific to any individual's employment situation. Readers should use their own judgment and, where appropriate, consult a qualified professional.

Published by Choral Software.

First edition, 2026.

Digital edition identifier: CHORAL_ALGORITHMIC_EDGE_EN_2026

Cover design: Choral Software, original digital artwork.

Dedication

For everyone who has stared down a blank editor window at the start of an interview and felt the clock start ticking — this book is for the version of you that walks out of the room knowing you gave it a real, structured shot.

Preface: Why Patterns Beat Memorization

Most people who prepare for coding interviews make the same mistake: they treat it as a memorization problem. They grind through hundreds of unrelated questions, hoping that sheer volume will cover whatever shows up on interview day. It rarely works, and when it does, it's exhausting and fragile — swap one detail in a familiar-looking question and the memorized solution collapses.

This book takes a different position. Underneath the enormous variety of coding interview questions sits a small, learnable set of patterns: two pointers, sliding windows, graph traversals, dynamic programming, and a dozen or so others. Once you can recognize which pattern a problem is really testing — often within the first thirty seconds of reading it — the rest of the work becomes far more mechanical. You're no longer solving a brand-new puzzle every time. You're applying a tool you've already practiced, to a slightly different shape of problem.

That's the promise of Algorithmic Edge: not another pile of disconnected questions to memorize, but a structured tour through the patterns themselves, with enough original practice problems attached to each one that recognizing the pattern becomes second nature. Every problem in this book was written specifically for it. None of them are lifted from any existing question bank, course, or book — they're original scenarios built to exercise the same underlying technique that shows up again and again in real interview loops.

Every solution in the book is worked through twice: once in Python 3 and once in Java. If you're a Python developer preparing for a Java-shop interview (or vice versa), or you simply want to see how the same idea looks in a dynamically typed, garbage-collected scripting language versus a statically typed, JIT-compiled one, that side-by-side view is deliberate. The two languages have different defaults and different failure modes, and understanding both makes you sharper in either one.

Read this book the way you'd approach the interviews themselves: methodically, pattern by pattern, testing yourself before you look at the answer. Good luck — and more importantly, good practice.

— Choral Software

How to Use This Book

Part I (Chapters 1-4) builds the foundation: how interviews are actually scored, how to reason about complexity, a refresher on the core data structures used everywhere else in the book, and a repeatable framework for approaching any new problem. If you're already comfortable with Big-O and the standard data structures, skim it — but don't skip Chapter 4, since its framework is referenced throughout the rest of the book.

Part II (Chapters 5-23) is the core of the book: nineteen chapters, each dedicated to a single pattern. Every chapter opens with the signals that tell you when the pattern applies, gives you a reusable code template in Python and Java, and then works through five to seven original problems of increasing difficulty — labeled Warm-up, Core, Stretch, or Capstone so you always know what level you're operating at. Work each problem yourself before reading the solution. That struggle is where the learning happens.

Part III (Chapters 24-27) covers material that shows up less often but disproportionately impacts your evaluation when it does: object-oriented design questions, segment and Fenwick trees, advanced dynamic programming, and concurrency. If your timeline is short, these chapters are the first candidates to deprioritize — but don't skip Chapter 24 if system-adjacent design questions are part of your target companies' loops.

Part IV (Chapters 28-30) shifts from what to solve to how to perform: communicating your thought process, five complete worked mock interviews, and guidance on building your own study schedule.

Suggested reading paths:

- Two weeks or less: Chapters 1, 4, then 5-8, 12, 13, 17-19, then Chapter 28 and one mock interview from Chapter 29 per remaining day.
- Six weeks: All of Part I, then Part II in order, spending roughly one day per chapter, then all of Part IV in the final week.
- Twelve weeks or more: The full book in order, including Part III, with a second pass through your weakest two or three chapters in the final two weeks. See Appendix C for a week-by-week template.

Every code sample in this book is written to run as-is in Python 3.10+ and Java 17+. Appendix A gives you a quick side-by-side syntax reference if you're translating your intuition from one language to the other.

Part I: Foundations

Big-O, the core data structures, and a repeatable framework for approaching any new problem.

Chapter 1: Understanding the Modern Technical Interview

Most candidates prepare for the wrong exam. They imagine that a technical interview is a math test in disguise — that somewhere out there is a specific, discoverable "right answer," and their job is to reproduce it as quickly as possible. That mental model produces a particular kind of anxiety: the fear of not knowing the trick. It also produces bad habits, like memorizing solutions to specific questions instead of building the underlying judgment that lets you construct a solution to a question you've never seen before.

The technical interview is not a math test. It is closer to a structured, hour-long observation session. An engineer sits across from you — or, more often now, across a video call and a shared code editor — and watches how you behave when handed a problem that doesn't have an obviously memorized answer. What they are measuring is not whether you land on the optimal solution. They are measuring whether you can be trusted to think clearly under pressure, communicate that thinking, and produce code that works. Those three things — clear thinking, clear communication, and working code — are the actual subject of the exam. Everything else in this book is in service of getting you good enough at those three things that a novel problem stops being scary.

This chapter has no code in it. Before you touch a single line of Python or Java, you need an accurate map of what's actually being evaluated, how the rounds of a real interview loop are typically structured, and what "good" looks like in practice. Getting this map right changes how you practice. Candidates who understand what's being scored practice differently — and more effectively — than candidates who are just trying to memorize more problems.

What interviewers are actually scoring

Every company runs its interviews a little differently, and most maintain some kind of structured scorecard behind the scenes. But strip away the company-specific templates and you'll find the same five dimensions showing up again and again, in some form, at nearly every organization that runs a real engineering interview loop.

Problem-solving process. This is the dimension candidates most underestimate. An interviewer wants to see you take an ambiguous prompt and turn it into a concrete, well-specified problem before you start typing. They want to see you consider more than one approach, reason out loud about trade-offs, and choose a direction deliberately rather than lunging at the first idea that comes to mind. A candidate who thinks for ninety silent seconds and then says, "I see two ways to approach this — a nested-loop check that's simple but quadratic, and a hash-based approach that trades some memory for linear time. Let me start with the second one, but I want to

make sure I understand the constraints first" is demonstrating exactly what this dimension is trying to measure. The specific data structure they land on matters less than the fact that they considered alternatives and made a reasoned choice.

Correctness. This is the dimension people over-index on, but it's still real. Does the code actually solve the stated problem? Does it handle the edges — an empty input, a single element, duplicate values, negative numbers, the largest allowed input size? A solution that works on the happy path but silently returns the wrong answer on an edge case is a serious red flag, because it's a preview of the kind of bug that reaches production. Interviewers are not looking for perfection on the first pass — nobody expects flawless code typed in real time under a stranger's gaze — but they are looking for a candidate who catches their own edge cases, either while writing the code or while testing it afterward.

Code quality. This dimension is about whether the code you produce under pressure resembles code a teammate would want to review. Are variable names meaningful, or are they `x`, `y`, `tmp2`? Is the logic broken into small, named pieces, or is it one sprawling block that nobody could safely modify six months from now? Do you handle obvious failure modes, or does the function assume its inputs are always well-formed? None of this needs to be perfect — interviews are not code-review sessions — but sloppy, unreadable code under low pressure is a strong signal of sloppy, unreadable code under real deadlines. Interviewers extrapolate.

Communication. This is the dimension most closely tied to how you'll actually function on a team, and it's scored constantly, not just at the end. Do you narrate your thinking as you work, so the interviewer can follow your reasoning instead of guessing at it? When the interviewer offers a nudge, do you take it gracefully and incorporate it, or do you get flustered and lose your thread? When you're stuck, do you say so and talk through what's blocking you, or do you go silent and hope the answer arrives? A candidate who talks through a wrong idea, catches the flaw themselves, and pivots is almost always scored higher than a silent candidate who eventually produces the same correct code — because the first candidate has demonstrated exactly the behavior their future teammates will rely on during a real debugging session.

Testing and verification. The final dimension is what you do after the code is "done." Do you trace through your own examples by hand, out loud, checking that the code produces the right output? Do you proactively raise edge cases the interviewer didn't mention? Or do you write the last line, sit back, and declare victory without checking anything? Strong candidates treat their own solution with a healthy amount of suspicion until they've proven to themselves — and to the interviewer — that it holds up. This single habit, more than any algorithmic trick, is what separates candidates who occasionally submit subtly broken code from those who don't.

Notice what's missing from this list: raw cleverness, encyclopedic knowledge of every algorithm ever devised, and typing speed. Those things can help at the margins, but they are not what the scorecard is built around. A candidate with a

modest but solid grasp of the core patterns, who communicates clearly and tests their work, will consistently outscore a candidate who knows more algorithms but freezes under pressure or writes code no one can follow.

How much weight goes to "optimal" versus "working"

This is the question candidates ask most often, and the honest answer is: it depends on the round, but working beats optimal far more often than most people assume.

A completely correct, well-tested, clearly explained solution at a suboptimal complexity — say, an $O(n^2)$ approach to a problem that has an $O(n \log n)$ solution — will, in most loops, score better than a broken or incomplete attempt at the optimal approach. Interviewers are trained to reward forward progress and honest reasoning. Nobody is impressed by a candidate who reaches for the cleverest possible algorithm, gets the implementation wrong under time pressure, and runs out of time debugging a subtle index error in an approach they didn't fully understand.

That said, complexity is not irrelevant, and treating "any working solution" as good enough is its own trap. Most interviewers expect you to notice when your first working approach is inefficient, say so out loud, and either improve it or explain clearly why you would improve it given more time. The sequence that tends to score best looks like this: get a correct brute-force approach working first, verify it against your examples, then explicitly discuss what's slow about it and why, and then — time permitting — implement the improvement. That sequence demonstrates every dimension on the scorecard at once: process (you considered the trade-off), correctness (you have a working baseline), communication (you narrated the improvement), and testing (you verified before and after). A candidate who jumps straight to a half-remembered optimal algorithm, skips the reasoning, and can't explain why it works is often scored worse than the candidate who walked calmly through the brute force first.

The practical implication for how you prepare: don't treat "know the optimal solution" as the finish line. Treat "reach a working solution methodically, then improve it deliberately" as the actual skill being tested. That is precisely the shape of the framework this book builds in Chapter 4 and reuses in every pattern chapter that follows.

The shape of a typical interview loop

Company processes vary in their specifics — some skip stages, some add extra ones, some compress everything into a single half-day — but a large majority of technical interview processes for software engineering roles follow a recognizable shape.

The loop usually opens with one or two screening rounds, most commonly conducted remotely over video with a shared, browser-based code editor rather than an actual whiteboard. A phone screen (the name persists even though it's rarely audio-only anymore) typically runs thirty to forty-five minutes and covers one, occasionally two, coding problems roughly in the "core" difficulty range you'll see throughout Part II of this book. The bar here isn't "impress the interviewer with brilliance" — it's "demonstrate baseline competence clearly enough that the company is willing to invest a full day of several engineers' time in you." Screening rounds are disproportionately about communication and process, precisely because the interviewer has the least time to gather signal and needs you to make your reasoning visible fast.

Candidates who clear the screen move to what's usually called the onsite loop, even when — as is now common — every round happens remotely. A typical onsite loop runs four to six rounds across a single day or split across a couple of days, each round forty-five minutes to an hour, each staffed by a different interviewer. The mix commonly includes: two or three rounds of data-structure-and-algorithm coding problems similar in spirit to the phone screen but often a notch harder or with a follow-up extension baked in; frequently a system-design or architecture round for anyone beyond the earliest career stages, where you're asked to design a larger piece of software at a whiteboard-diagram level rather than write code; often a round focused on behavioral and collaboration questions — how you've handled disagreement, ambiguity, or failure on real projects; and, depending on the team, a round that blends object-oriented design with coding, where you build a small class-based system live rather than solving a self-contained algorithmic puzzle (Chapter 24 covers this format directly).

Within any individual coding round, you'll typically encounter one of three formats. The shared-editor format — a browser-based code editor, sometimes with syntax highlighting and even a runnable console, sometimes without — is by far the most common today, especially for remote loops. The live-pairing format, more common at companies that want to simulate real day-to-day work, has you and the interviewer collaborating more actively, sometimes with the interviewer occasionally suggesting a direction or asking you to extend the problem as you go. The whiteboard format, once the industry default, has become less common for early rounds but still shows up, particularly for in-person onsite loops or for rounds explicitly designed to test how you reason without the safety net of a compiler catching your syntax errors. Each format rewards slightly different micro-skills — whiteboard rounds put more weight on getting your logic right without machine verification, shared-editor rounds put more weight on writing genuinely runnable code — but the underlying scorecard dimensions from the previous section apply identically across all three. Practicing exclusively in one format is a common mistake; if you know your upcoming loop uses a whiteboard round, spend at least some practice time away from an IDE's autocomplete and instant feedback.

It's worth naming explicitly what most loops are not, because the myth persists: it is rare, at any reputable company, for a single question to have a single "password" answer that either unlocks a hire or doesn't. Interviewers are typically trained to evaluate the dimensions above holistically, across the whole round, and a debrief across the full loop weighs signal from every round together. One rough round rarely sinks a candidate who performed well elsewhere and can articulate what went wrong; one lucky guess on a single trick rarely carries a candidate whose reasoning was weak everywhere else.

A worked narrative: two responses to the same prompt

To make all of this concrete, consider a single generic prompt — deliberately unremarkable, because the point isn't the specific problem, it's the behavior — and two different ways a candidate might respond to it.

The prompt: "You're given a list of integers representing hourly sensor readings from a single day. Write a function that returns the length of the longest stretch of consecutive hours during which the readings were strictly increasing."

A weaker response starts almost immediately. The candidate reads the prompt once, nods, and starts typing a nested loop: for each starting hour, walk forward as far as the readings keep increasing, track the longest run seen. They don't ask about empty input. They don't ask what should happen if no two consecutive hours increase. They type in silence for close to two minutes, narrate nothing, then present a solution. When the interviewer asks, "What's the time complexity of this?" the candidate pauses, unsure, and eventually guesses "O(n), I think?" — incorrectly, since the nested walk is actually $O(n^2)$ in the worst case (a strictly increasing array of length n forces the inner loop to walk the full remaining length from nearly every starting point). When the interviewer asks what happens with an empty list, the candidate traces through it mentally, realizes the function would throw an error on an unguarded access, and patches it defensively without fully explaining why. The final code works on the examples given, but the interviewer leaves the round uncertain whether the candidate understood their own solution's cost, and uncertain how the candidate handles unanticipated edge cases when not explicitly prompted.

A stronger response to the identical prompt looks different from the first ten seconds. The candidate restates the problem in their own words — "so I want the longest run of indices where each reading is strictly greater than the one before it, and I should return the length of that run, not the run itself, correct?" — and gets a quick confirmation. They ask two clarifying questions: what should the function return for an empty list (the interviewer says zero), and does "strictly increasing" mean no ties count as part of a run (confirmed: ties break the run). They jot two quick examples by hand, including one with a tie and one that's a single element, and state the expected output for each before writing any code. Then they think out loud for a moment: "I could compare every pair of indices, but I think I can do this in a single

pass — I just need to track the length of the current increasing run and the best one seen so far, resetting the current run whenever the sequence stops increasing." They write roughly eight lines of clean, well-named code, narrating each piece as they go. When they finish, they don't declare victory — they trace both handwritten examples through the code line by line, catch that their reset condition was off by one hour in the tie case, fix it on the spot, and re-trace to confirm. Only then do they state the complexity: "single pass, so O(n) time, and O(1) extra space since I'm just tracking two counters." The interviewer asks a follow-up — "what if I wanted the longest run of non-decreasing readings instead?" — and the candidate talks through the one-line change needed without touching the keyboard, since they can see exactly which comparison controls that behavior.

Both candidates, notice, arrive at correct code by the end. That's precisely why this example matters: correctness alone did not separate them. What separated them was process (restating the problem and planning before typing), communication (narrating decisions instead of leaving the interviewer to infer them), testing (tracing hand-built examples and catching a real bug before being asked to), and the ability to reason about complexity and extensions on demand rather than guessing. An interviewer scoring these two rounds is not comparing two pieces of final code side by side — they are comparing two demonstrations of engineering judgment, and only one of those demonstrations gives them confidence about how the candidate will behave on a real team, on a real problem, with real ambiguity and no example test cases handed to them in advance.

That is the exam you're actually studying for. The rest of this book exists to make the stronger response above your default mode — not through memorization, but by giving you a durable framework for approaching new problems (Chapter 4), a fluent grasp of the data structures and complexity reasoning that make quick, confident trade-off decisions possible (Chapters 2 and 3), and enough deliberate practice recognizing the patterns underneath a huge swath of interview problems (Part II onward) that the "planning" phase of a new problem stops feeling like starting from zero every single time.

Chapter 2: Big-O and Complexity Analysis, Properly

Every solution in this book ends with a one-line complexity statement, and every interviewer you sit in front of will ask you to produce one, unprompted, the moment your code compiles in their head. Complexity analysis is the shared language that lets you and an interviewer agree, quickly and precisely, on how a piece of code will behave as its input grows — without either of you needing to run it. Treat this chapter as more than a formality. A shaky grasp of Big-O is one of the most common reasons otherwise-solid candidates lose points, because it shows up not just as a single question at the end of a round but as a constant undertone to every decision you make while coding: which data structure to reach for, whether a nested loop is a red flag, whether a recursive approach will blow the call stack.

The goal here is to build real fluency, not just the ability to recite "nested loop equals order n squared." You should walk out of this chapter able to look at code you've never seen before — including recursive code, code with hidden costs, and code whose apparent nested loops don't actually multiply — and derive its complexity from first principles rather than pattern-matching on shape alone.

What Big-O actually means

Big-O notation describes how the cost of an algorithm — usually time, sometimes space — grows as the size of its input, conventionally called `n`, grows without bound. It deliberately throws away two kinds of detail: constant factors (whether an operation takes one nanosecond or one microsecond) and lower-order terms (a `+ 5` or a `+ n` tacked onto a dominant n^2 term). What's left is a description of shape — does the cost grow linearly, quadratically, logarithmically — rather than a precise measurement. That's a feature, not a sloppiness: shape is what determines whether your code still runs in a reasonable time when the input grows from a thousand elements to a hundred million, and constant factors mostly don't matter at that scale.

Formally, a function `f(n)` is $O(g(n))$ if there exist positive constants `c` and n_0 such that $f(n) \le c \cdot g(n)$ for every $n \ge n_0$. In plain language: once the input is large enough, `f(n)` never exceeds some constant multiple of `g(n)`. Big-O is an upper bound — it says the algorithm is no worse than a certain growth rate, but it doesn't promise that growth rate is tight. A linear-time algorithm is technically also $O(n^2)$, $O(n^3)$, and $O(2^n)$, because all of those are valid upper bounds; we just don't say that in practice, because it's an unhelpfully loose bound. When people say "this is O(n)" in a conversation, they almost always mean the tightest commonly-used upper bound, and interviewers expect that convention.

Big-Omega (Ω) is the mirror image: a lower bound. `f(n)` is $\Omega(g(n))$ if `f(n)` never falls below some constant multiple of `g(n)` for large enough `n`. Big-Theta (Θ) is

what you get when the upper and lower bounds match — `f(n)` is `Θ(g(n))` if it's both `O(g(n))` and `Ω(g(n))`, meaning `g(n)` describes the exact asymptotic growth rate, not just a ceiling or a floor. In interview conversation, people are frequently loose about this and say "Big-O" when they mean "Theta" — when someone says an algorithm "is O(n log n)," they usually mean it genuinely grows like `n log n`, not merely that it's no worse than `n log n`. You don't need to correct anyone's terminology out loud, but understanding the distinction sharpens your own reasoning, particularly when an algorithm's worst case and typical case differ (a hash map lookup is `Θ(1)` on average but only `O(n)` in the worst case — those are two different, both-true statements about the same operation, and knowing which one an interviewer is asking about matters).

Reading complexity out of loops

The most common way complexity shows up in interview code is through loops, and the mechanical rule is straightforward once you internalize it: count how many total iterations happen, across all loops, as a function of the input size.

A single loop that runs once per element of an `n`-element input is `O(n)`. Two sequential loops — one after the other, each independently touching the input once — are still `O(n)`, because you add their costs rather than multiply them: `O(n) + O(n) = O(2n)`, and constant factors drop out, leaving `O(n)`. This trips people up more than it should; a candidate who sees two separate loops in a row and reflexively says "O(n²)" is making a real mistake, and it's worth training yourself out of that reflex specifically.

Nested loops are where multiplication enters. If an outer loop runs `n` times and, for each outer iteration, an inner loop also runs on the order of `n` times, the total work is proportional to `n × n = n²`. This is the classic quadratic pattern — checking every pair of elements in a list, for instance. If the inner loop's range shrinks as the outer index grows (a common pattern in triangle-shaped iteration, like comparing each element only to the elements after it), the total number of iterations is still proportional to `n²` — the sum `1 + 2 + ... + n` is `n(n+1)/2`, and the leading term is still quadratic once constants are dropped. Three independently nested loops give you `O(n³)`, and so on; each additional level of loop nesting over the same input generally multiplies another factor of `n` onto the total.

Not every loop that looks nested is actually multiplicative, though, and this is where mechanical pattern-matching fails candidates who haven't built real intuition. Consider a two-pointer scan across an array, where an outer `while` loop advances one pointer and an inner `while` loop advances a second pointer, but crucially, neither pointer ever moves backward, and each can move forward at most `n` times over the life of the whole function — not per outer iteration, but in total. In that shape, even though the code has the visual structure of a nested loop, the total number of inner-loop steps across the entire run is bounded by `n`, not by `n` per outer step. That's `O(n)`, not `O(n²)`, and correctly identifying that distinction is a frequent,

high-value "aha" moment in interviews — you'll see this exact reasoning applied concretely in the two-pointer and sliding-window chapters later in the book.

Recursion and the recursion tree

Recursive code doesn't have "loops" in the textbook sense, but it has an analogous mechanical technique: draw (or imagine) the tree of recursive calls, and count the total work across every node in that tree.

For a recursive function that makes one recursive call, each call handling a problem of size roughly `n - 1` (or some other simple reduction) and doing `O(1)` work outside the recursive call, the recursion tree is a single chain of depth `n`, and the total work is `O(n)` — same as an equivalent loop, because it essentially is an equivalent loop, just expressed with call-stack frames instead of a `for` statement.

Where recursion gets interesting is branching recursion — a function that makes more than one recursive call per invocation. A function that calls itself twice, each call handling a problem of size `n - 1`, produces a recursion tree that's a full binary tree of depth `n`. A binary tree of depth `n` has roughly 2^n leaf nodes, and since work happens at (or near) every node, the total cost is $O(2^n)$ — exponential. This is precisely the trap in naive, non-memoized recursive solutions to problems like counting the number of ways to climb a staircase using variable step sizes, or computing terms of a sequence defined by summing two previous terms: the recurrence looks innocent, but unrolled without any caching, it explodes combinatorially, doing enormous amounts of redundant repeated work.

Recursion that halves the problem at each step — binary search is the canonical example — produces a very different shape: a recursion tree that's a single chain, but where the input size divides by two at each level rather than decreasing by one. A quantity that starts at `n` and halves repeatedly reaches `1` after roughly $\log n$ steps, so the recursion has depth $O(\log n)$, and if each level does $O(1)$ work, the total cost is $O(\log n)$ — logarithmic, dramatically cheaper than linear once `n` gets large. (An input of a billion elements needs only about thirty halving steps to whittle down to one.)

The Master Theorem, intuitively

Divide-and-conquer recursion — the family that includes efficient sorting algorithms like merge sort, and many of the "split the problem, solve the pieces, combine" strategies you'll meet in later chapters — has a general recurrence shape: a problem of size `n` is split into `a` subproblems, each of size `n / b`, and the results are combined with $O(n^d)$ extra work outside the recursive calls. The Master Theorem is a shortcut for solving that shape of recurrence without redoing the recursion-tree analysis by hand every time, and you don't need to memorize its algebraic form to use its intuition productively in an interview.

The intuitive version comes down to a race between two quantities: how fast the number of subproblems grows as you go down the tree (driven by `a` and `b`) versus how much work is done per node combining results (driven by `d`). If the combining work per level dominates — the amount of work done at the top of the tree, before any splitting, is already comparable to or bigger than everything below it — the total cost is dominated by that top-level work, $O(n^d)$. If instead the sheer number of leaf subproblems dominates — splitting produces so many small pieces that their combined base-case cost outweighs the combining work — the total cost is dominated by the leaves. And if the two forces are perfectly balanced, you get an extra logarithmic factor tacked onto the shared rate, which is exactly why merge sort, splitting an array in half at every level and doing $O(n)$ work to merge the halves back together, lands at $O(n \log n)$ rather than plain $O(n)$ or plain $O(n^2)$: the per-level combining work stays proportional to n at every one of the $O(\log n)$ levels, and those levels add rather than multiply. You'll see this reasoning made concrete, rather than abstract, when Part II covers algorithms that are explicitly built on a divide-and-conquer skeleton.

Amortized analysis: the dynamic array, unrolled

Some operations are expensive occasionally and cheap the rest of the time, and describing their cost accurately requires looking at a whole sequence of operations rather than any single one in isolation. The clearest example, and the one worth internalizing deeply because it recurs constantly, is appending to a dynamic array — Python's `list`, Java's `ArrayList`.

A dynamic array is backed by a fixed-size block of memory. Appending an element is normally $O(1)$: write the value into the next open slot, bump a size counter. But when that backing block is full, appending requires allocating a new, larger block — typically double the size — and copying every existing element into it, which is an $O(n)$ operation for an array that currently holds n elements. Taken in isolation, that resize step looks like it makes "append" an $O(n)$ operation in the worst case, and if you only look at the single append that triggers a resize, that's true.

But resizes don't happen often, and doubling the capacity each time is precisely what keeps them rare. Consider building an array up to n elements from empty, doubling capacity every time it fills: resizes happen when the array holds roughly $1, 2, 4, 8, 16, \ldots$ elements — a set of sizes that itself only has about $\log n$ entries. The total cost of all the resize-triggered copying across the entire sequence of n appends is $1 + 2 + 4 + \ldots + n$, a geometric series that sums to roughly $2n$ — linear in n, not $n \log n$ and certainly not n^2, even though individual append calls occasionally cost $O(n)$. Spread that total $O(n)$ cost evenly across the n append calls that produced it, and each append costs $O(1)$ on average, or "amortized." That's the precise meaning of amortized $O(1)$: not that every single call is cheap, but that the total cost of any long sequence of calls, divided by the number of calls, converges to a constant. This is exactly why Python's `list.append` and

Java's `ArrayList.add` are both routinely described as `O(1)` despite the occasional expensive resize under the hood, and why doubling — rather than, say, growing by a fixed amount of ten elements each time — is the specific choice that makes the amortized argument work; growing by a fixed constant instead of a fixed fraction would make resizes happen every `O(1)` appends indefinitely, pushing the amortized cost back up to `O(n)` per append. That distinction is worth being able to explain out loud — it's a favorite interviewer follow-up whenever a dynamic array comes up.

Space complexity, including the call stack

Space complexity follows the same growth-rate logic as time complexity, but measures memory instead of operations, and candidates consistently forget one major contributor: the call stack used by recursion.

Every active recursive call consumes a stack frame — space to hold that call's local variables and its return address — and that frame stays allocated for as long as the call (and everything it calls) hasn't yet returned. A recursive function with a single chain of depth `n`, like the naive linear recursion described earlier, uses `O(n)` auxiliary space for the call stack alone, even if it doesn't allocate a single explicit data structure — a fact that surprises candidates who mentally credit recursive solutions with "no extra space" just because they don't see an array or a hash map anywhere in the code. A recursive function that halves its input at each step, by contrast, only ever has `O(log n)` frames alive at once, because the chain of nested calls is only `O(log n)` deep even though the total number of calls across the whole tree might be larger (in a branching recursion, total calls and maximum simultaneous depth are different quantities, and it's depth — not the total call count — that determines stack space). When you state the space complexity of a recursive solution in an interview, always account for the call stack explicitly, and say so out loud; it's an easy, high-signal detail to include and an easy one to be caught skipping.

Exercise set: estimate the complexity

Below are six short, original Python snippets. For each, work out the time complexity yourself — say it out loud, the way you would in an interview — before reading the explanation underneath. Space complexity is discussed where it's the interesting part of the answer.

Snippet 1

python

```python
def has_matching_pair(amounts, target):
    n = len(amounts)
    for i in range(n):
        for j in range(i + 1, n):
            if amounts[i] + amounts[j] == target:
                return True
    return False
```

Answer: O(n²) time, O(1) auxiliary space. The outer loop runs `n` times, and for each outer iteration the inner loop runs a shrinking but still linear number of times (`n - i - 1` iterations). Summing `1 + 2 + ... + (n-1)` gives a total proportional to n^2 once constants are dropped — the classic all-pairs comparison. No auxiliary data structure grows with input size, so space is constant, ignoring the input itself.

Snippet 2

python

```
def find_insert_slot(prices, target):
    low, high = 0, len(prices)
    while low < high:
        mid = (low + high) // 2
        if prices[mid] < target:
            low = mid + 1
        else:
            high = mid
    return low
```

Answer: O(log n) time, O(1) space. This is a binary search variant: each iteration discards half of the remaining `[low, high)` range, so the range shrinks from `n` to `0` in about `log n` steps. No recursion is used here, so there's no call-stack cost either — just two integer variables, hence constant space.

Snippet 3

python

```
def has_common_value(sorted_a, sorted_b):
    i, j = 0, 0
    while i < len(sorted_a) and j < len(sorted_b):
        if sorted_a[i] == sorted_b[j]:
            return True
        elif sorted_a[i] < sorted_b[j]:
            i += 1
        else:
            j += 1
    return False
```

Answer: O(n + m) time, O(1) space, where `n` and `m` are the lengths of the two input arrays. This looks superficially like it could be quadratic because there's a loop with two index variables being advanced, but look closely: on every single iteration of the `while` loop, at least one of `i` or `j` increases by one, and neither ever decreases or resets. Since `i` can advance at most `n` times total and `j` can advance at most `m` times total over the entire run of the function — not per iteration of some outer loop, but across the whole call — the loop body can execute at most `n + m` times before one of the two index variables runs out of room. This is the two-pointer shape flagged in the loop-reading section above: it has the visual texture of something that might be quadratic, but the total work is strictly linear in the combined input size.

Snippet 4

python

```
def count_step_paths(n):
    if n <= 1:
        return 1
    return count_step_paths(n - 1) + count_step_paths(n - 2)
```

Answer: O(2ⁿ) time, O(n) space. Each call (for `n > 1`) makes two recursive calls, so the recursion tree branches in two directions at every node down to a depth of roughly `n`, giving an exponential total call count — a direct instance of the branching-recursion trap described earlier, and a textbook argument for adding memoization (caching each `n` the first time it's computed) to bring this down to `O(n)` time, which Chapter 18 covers as a fully worked pattern. The space complexity is not exponential, though, and this is the detail candidates most often get wrong: at any given moment, only one path down the recursion tree is "active" on the call stack — the tree branches, but the branches are explored one at a time, sequentially, not simultaneously — so the maximum call-stack depth is `O(n)`, not `O(2ⁿ)`.

Snippet 5

python

```python
def build_receipt(item_names):
    receipt = ""
    for name in item_names:
        receipt += name + ", "
    return receipt
```

Answer: O(n²) time in the worst case (assuming item names are roughly constant length), O(n) auxiliary space for the final string. This is a hidden-cost trap rather than a visibly nested loop. Strings in Python are immutable, so each `receipt += ...` doesn't extend the existing string in place — it allocates an entirely new string and copies the old contents into it, character by character, plus the new piece. If the loop runs `n` times and the receipt string grows roughly linearly in length with each iteration, the total copying work across all iterations sums to `1 + 2 + ... + n`, which is `O(n²)`, for exactly the same reason the all-pairs comparison in Snippet 1 was quadratic — a linear number of steps, each doing linearly growing work. The fix, when this pattern shows up for real, is to accumulate pieces in a list and join them once at the end (or use Java's `StringBuilder`, which is built precisely to avoid this trap), turning the total cost back into `O(n)`.

Snippet 6

python

```python
def answer_range_queries(records, queries):
    results = []
    for lo, hi in queries:
        window = sorted(records[lo:hi])
        results.append(window[len(window) // 2])
    return results
```

Answer: O(q · n log n) time in the worst case, where `q` is the number of queries and `n` is the length of `records` (more precisely, if `k` is the length of the slice for a given query, that query costs `O(k log k)`; the stated bound is the worst case where each slice is close to the full array). The loop itself runs `q` times, which by itself would only suggest `O(q)` — but each iteration doesn't do constant work: it slices out a sub-list and sorts it from scratch, and sorting a `k`-element list costs `O(k log k)`. Multiplying a linear count of iterations by a non-constant per-iteration cost is a second common hidden-cost trap, distinct from Snippet 5's version: here the giveaway isn't string

immutability, it's a function call — `sorted(...)` — that looks like a single cheap statement but actually hides a non-trivial algorithm inside it. Whenever you see a sort, a search, or any other non-trivial built-in call sitting inside a loop, stop and ask what that call costs on its own before declaring the whole function's complexity.

Bringing it into the interview room

You won't be asked to recite the formal definition of Big-O in a real interview, but you will be expected to produce a correct, justified complexity statement for your own code within a few seconds of finishing it — and, just as often, to reason about the complexity of an approach you haven't written yet, while you're still deciding which approach to pursue. That second skill is the one worth building most deliberately: before you write a line of code for any problem in this book, get in the habit of stating out loud what you expect the time and space complexity to be, based on the shape of the approach alone. Then verify that prediction once the code exists. If they disagree, figure out why before moving on — that gap is almost always teaching you something you missed, whether it's a hidden loop, an unaccounted-for call-stack cost, or an amortized effect you didn't consider. This habit, more than memorizing growth-rate labels, is what makes complexity analysis feel automatic by the time it matters.

Chapter 3: Data Structures Refresher

Every pattern in this book is built on top of a small set of data structures, and the fluency of your solutions during an interview depends almost entirely on how automatically you can reach for the right one. When a candidate freezes mid-interview, it's rarely because they don't understand the algorithm — it's because they're simultaneously trying to remember how a hash set handles collisions, or whether removing from the front of a Python list is cheap, while also trying to reason about the problem itself. This chapter exists to clear that second layer out of the way permanently.

This is a refresher, not a first introduction — if terms like "linked list" or "binary search tree" are completely new to you, a slower, more foundational resource will serve you better before returning here. What this chapter does provide is a fast, precise pass through every structure you'll lean on for the rest of the book, with the operations and complexities stated explicitly enough that you can use this chapter as a lookup reference during your practice sessions. Each section includes a short, runnable snippet in both Python and Java showing the structure's core operations in action, because the fastest way to re-solidify "how expensive is this operation, really" is to see it used against a small, concrete example rather than read about it in the abstract.

One framing to hold onto throughout: almost every data structure decision in an interview is a trade-off between how fast you can find something, how fast you can change something, and how much memory you're willing to spend to get there. Keep that three-way tension in mind as you read, because nearly every structure below is best understood as a specific, deliberate answer to it.

Arrays and strings

An array is a block of memory holding elements of the same type, laid out contiguously, so that any element can be reached directly by computing an offset from the start of the block. That contiguity is the entire reason arrays are fast to read from: accessing `array[7]` doesn't require walking through the first seven elements, it requires one arithmetic calculation. Strings, in most languages you'll use in an interview, are effectively arrays of characters with one additional wrinkle worth knowing cold: in Python, strings are immutable, so any operation that appears to "modify" a string actually builds an entirely new one, and in Java, `String` is likewise immutable, which is precisely why `StringBuilder` exists as the mutable alternative when you're constructing a string piece by piece in a loop (Snippet 5 in Chapter 2 walked through exactly this cost trap).

The operations worth having complexity intuition for: reading or writing at a known index is `O(1)`, because of that direct-offset property. Searching for a value you don't know the position of is `O(n)`, because in the general unsorted case there's no shortcut — you have to look at elements until you find it or run out. Inserting or removing at the end of an array is typically cheap, but inserting or removing anywhere else requires shifting every subsequent element over by one slot to keep the array contiguous, which costs `O(n)` in the worst case.

python

```python
readings = [68, 71, 69, 75]
readings.append(73)          # O(1) amortized
readings.insert(0, 66)       # O(n): shifts every element right
middle = readings[3]         # O(1): direct index
readings.remove(69)          # O(n): search, then shift left
print(readings)
```

java

```java
List<Integer> readings = new ArrayList<>(List.of(68, 71, 69, 75));
readings.add(73);                      // O(1) amortized
readings.add(0, 66);                   // O(n): shifts every element right
int middle = readings.get(3);          // O(1): direct index
readings.remove(Integer.valueOf(69)); // O(n): search, then shift left
System.out.println(readings);
```

A detail that trips people up in Java specifically: `readings.remove(69)` on a `List<Integer>` would remove the element at index 69, not the value `69`, because `remove(int)` is overloaded separately from `remove(Object)`. Wrapping the value in `Integer.valueOf(69)` forces Java to resolve the call to the object-removal overload. It's a small syntactic trap, but it's exactly the kind of thing that costs a candidate visible confusion mid-interview if they haven't seen it before.

Strings deserve one more concrete habit worth naming: slicing. Python's `s[2:5]` and Java's `s.substring(2, 5)` both produce a new string, and both cost time proportional to the length of the slice being produced, not the length of the original string — a fact candidates often assume is "free" because the syntax is so lightweight. When a problem statement mentions strings that could be very long (log lines, document text, DNA sequences), it's worth explicitly noting out loud whether your approach repeatedly slices a large string inside a loop, because that pattern quietly turns an otherwise linear algorithm quadratic in the same way string concatenation does. The fix is usually the same one used for concatenation: work with indices into the original string rather than materializing new substrings until you actually need to return one.

Dynamic arrays

Python's `list` and Java's `ArrayList` are both dynamic arrays: arrays that automatically grow their backing storage when they run out of room, rather than requiring you to declare a fixed size up front. Chapter 2 walked through exactly why this makes appending an amortized `O(1)` operation despite the occasional expensive resize — the backing array doubles in size whenever it fills, and that doubling keeps the total cost of all resizes across a long sequence of appends linear rather than quadratic. It's worth seeing that mechanism made concrete once, rather than only in the abstract, because "amortized" stops being a slippery word once you've watched a capacity variable double in front of you.

python

```
class DynamicArray:
    def __init__(self):
        self._capacity = 1
        self._size = 0
        self._store = [None] * self._capacity

    def append(self, value):
        if self._size == self._capacity:
            self._capacity *= 2
            bigger = [None] * self._capacity
            for i in range(self._size):
                bigger[i] = self._store[i]
            self._store = bigger
        self._store[self._size] = value
        self._size += 1

    def __getitem__(self, index):
        if index >= self._size:
            raise IndexError(index)
        return self._store[index]

arr = DynamicArray()
for value in range(10):
    arr.append(value * value)
print([arr[i] for i in range(10)], arr._capacity)
```

java

```
class DynamicArray {
    private int[] store;
    private int size = 0;
    private int capacity = 1;

    DynamicArray() {
        store = new int[capacity];
    }

    void append(int value) {
        if (size == capacity) {
            capacity *= 2;
            int[] bigger = new int[capacity];
            System.arraycopy(store, 0, bigger, 0, size);
            store = bigger;
        }
        store[size] = value;
        size++;
    }

    int get(int index) {
        if (index >= size) throw new IndexOutOfBoundsException(Integer.toString(index));
        return store[index];
    }
}
```

Running the Python version and appending ten elements to a structure that started with capacity `1` leaves the backing array at capacity `16` — it doubled at sizes `1, 2, 4, 8` — having wasted a few unused slots in exchange for never having paid for more than `O(n)` total copying across the whole sequence. That wasted space is the concrete price of amortized `O(1)` appends, and it's a fair trade in nearly every practical situation. In an interview, you're extremely unlikely to be asked to implement a dynamic array from scratch, but you will be expected to know, instantly and without hesitation, that `list.append` in Python and `ArrayList.add` in Java are both amortized `O(1)`, and that inserting at the front of either is `O(n)`.

Two related facts are worth having ready as well. First, if you know in advance roughly how many elements you're going to store, pre-sizing the structure — `ArrayList<>(expectedSize)` in Java, or simply building a Python list via a comprehension instead of a loop of individual `append` calls — avoids the resize churn entirely and is a small, genuine performance habit worth mentioning if a problem's constraints give you a firm upper bound. Second, shrinking is asymmetric with growing: neither Python's `list` nor Java's `ArrayList` automatically shrinks its backing storage when you remove elements, which is a deliberate choice, since a structure that shrank as eagerly as it grows would risk oscillating between growing and shrinking on alternating append/remove calls, destroying the amortized guarantee entirely. If you ever do need to reclaim memory after removing a large number of elements, that has to be done explicitly (`list.trimToSize()` in Java has no true Python equivalent, since Python's memory model works differently under the hood) — it's a minor point, but naming it correctly if asked signals real fluency rather than surface familiarity.

Singly linked lists

A singly linked list stores elements as a chain of nodes, each holding a value and a reference to the next node, with no requirement that the nodes sit anywhere near each other in memory. That gives up the direct-offset indexing that makes arrays fast to read, but it buys something arrays can't offer cheaply: inserting or removing a node at a position you already have a reference to is `O(1)`, because it only requires rewiring a couple of pointers, not shifting any other elements.

python

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def push_front(self, value):
        self.head = Node(value, self.head)   # O(1)

    def find(self, value):
        current = self.head
        while current:                       # O(n)
            if current.value == value:
```

```
                return current
            current = current.next
        return None

hops = SinglyLinkedList()
for host in ["gateway", "core-3", "core-1", "edge-9"]:
    hops.push_front(host)
print(hops.find("core-1").value)
```

java

```
class Node {
    String value;
    Node next;
    Node(String value, Node next) { this.value = value; this.next = next; }
}

class SinglyLinkedList {
    Node head;

    void pushFront(String value) {
        head = new Node(value, head);   // O(1)
    }

    Node find(String value) {
        Node current = head;
        while (current != null) {       // O(n)
            if (current.value.equals(value)) return current;
            current = current.next;
        }
        return null;
    }
}
```

Notice the Java `find` method uses `.equals(value)`, not `==`, to compare strings — `==` on Java objects checks reference identity, not content equality, and using it here would give wrong answers for strings that hold the same characters but aren't the same object in memory. This is one of the sharpest-edged differences between Python and Java that shows up constantly in interview code; Appendix A covers it in more depth. Reading the value at a known position (the fifth node, say) is `O(n)` in a singly linked list, because there's no shortcut to the fifth node other than walking there one link at a time — a real cost that arrays don't have, and the central trade-off to weigh whenever you're deciding between the two structures.

A pattern worth internalizing now, because it recurs across nearly every linked-list problem in Chapter 10, is the dummy head node: a placeholder node created before the real list, whose `next` pointer is set to the actual head, used purely to simplify edge cases in code that inserts or removes near the front of the list. Without a dummy head, code that might delete the very first node needs a separate branch just for that case (there's no "previous" node to relink); with one, every deletion — first node included — follows exactly the same relinking logic, because the dummy always plays the role of "the node before this one." It's a small trick, but it removes a disproportionate share of the off-by-one and null-pointer bugs that show up in live-coded linked-list solutions, and reaching for it early, before you've written five lines of special-casing, is a habit worth building now rather than mid-interview.

Doubly linked lists

A doubly linked list extends the same idea with a second pointer per node, pointing back to the previous node as well as forward to the next. The extra pointer costs a little memory per node, but it buys two things that matter constantly in interview problems: you can walk the list in either direction, and — critically — if you already hold a reference to some node in the middle of the list, you can remove that node in `O(1)`, without needing to first walk from the head to find its predecessor (a singly linked list would need `O(n)` just to locate the node before it, unless you happened to already have that reference too).

python

```
class DNode:
    def __init__(self, value, prev=None, next=None):
        self.value = value
        self.prev = prev
        self.next = next

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def push_back(self, value):
        node = DNode(value, prev=self.tail)
        if self.tail:
            self.tail.next = node
        else:
            self.head = node
        self.tail = node                      # O(1)

    def remove(self, node):
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next
        if node.next:
            node.next.prev = node.prev
        else:
            self.tail = node.prev             # O(1) given the node reference
```

java

```
class DNode {
    String value;
    DNode prev, next;
    DNode(String value, DNode prev) { this.value = value; this.prev = prev; }
}

class DoublyLinkedList {
    DNode head, tail;

    DNode pushBack(String value) {
        DNode node = new DNode(value, tail);
        if (tail != null) tail.next = node; else head = node;
        tail = node;                          // O(1)
        return node;
    }

    void remove(DNode node) {
        if (node.prev != null) node.prev.next = node.next; else head = node.next;
        if (node.next != null) node.next.prev = node.prev; else tail = node.prev; // O(1)
    }
}
```

This `O(1)`-removal-given-a-reference property is precisely why doubly linked lists show up as a building block inside more elaborate structures later in the book — an LRU cache, for instance, pairs a doubly linked list with a hash map specifically so it can locate a node instantly (via the map) and then evict it instantly (via the list), a combination you'll build in full in Chapter 24.

It's worth being explicit about when a doubly linked list is not the right call, since candidates sometimes reach for one reflexively once they know the trick. If your access pattern is dominated by reading elements at arbitrary positions rather than inserting and removing given a reference you already hold, a plain dynamic array is almost always faster in practice, despite the linked list's better big-O for insertion — the array's contiguous memory layout plays far better with modern CPU caches, and the constant-factor difference is large enough to matter outside of asymptotic analysis. Big-O tells you how cost scales; it doesn't tell you which structure wins for the sizes you'll actually see, and it's a good habit to say so out loud when a problem's constraints suggest the input will stay small.

Stacks and queues, including the deque

A stack is a last-in-first-out structure: the most recently added element is the first one removed. A queue is first-in-first-out: the oldest added element is the first removed. Both can be implemented on top of either an array or a linked list, and both support their core operations — push/pop for a stack, enqueue/dequeue for a queue — in `O(1)`, provided you pick an implementation that avoids shifting elements (adding to and removing from the end of a dynamic array is cheap, which is why a dynamic array makes a perfectly good stack; a naive array-based queue that always removes from index `0` would cost `O(n)` per dequeue unless it's specifically designed to avoid that).

A deque (double-ended queue) generalizes both: it supports `O(1)` addition and removal from both ends. Python's `collections.deque` and Java's `ArrayDeque` are both implemented so that every one of those four operations — push front, push back, pop front, pop back — is `O(1)`, which is why a deque is the default choice in this book whenever a problem needs stack-like behavior, queue-like behavior, or both at different points in the same algorithm (the sliding-window and monotonic-stack chapters lean on this constantly).

python

```
from collections import deque

undo_stack = []
undo_stack.append("draw-circle")   # O(1)
undo_stack.append("move-shape")
last_action = undo_stack.pop()     # O(1)

print_queue = deque()
print_queue.append("job-14")       # O(1) enqueue at right
print_queue.append("job-15")
next_job = print_queue.popleft()   # O(1) dequeue at left
print(last_action, next_job, list(print_queue))
```

java

```
Deque<String> undoStack = new ArrayDeque<>();
undoStack.push("draw-circle");   // O(1)
undoStack.push("move-shape");
String lastAction = undoStack.pop(); // O(1)

Deque<String> printQueue = new ArrayDeque<>();
printQueue.addLast("job-14");    // O(1) enqueue at tail
printQueue.addLast("job-15");
String nextJob = printQueue.pollFirst(); // O(1) dequeue at head
System.out.println(lastAction + " " + nextJob + " " + printQueue);
```

Java doesn't have separate `Stack`-flavored and `Queue`-flavored classes for everyday use the way the naming might suggest — `ArrayDeque` plays both roles, and is generally preferred over the legacy `java.util.Stack` class, which is built on older, synchronized, and slower foundations. Python's plain `list` works fine as a stack (`append`/`pop` from the end are both `O(1)`), but should not be used as a queue via `pop(0)`, which is `O(n)` — `collections.deque` is the correct tool the moment you need cheap removal from the front.

The bounded-history example in the snippet above — capping a deque at a fixed size so old entries fall off automatically once it's full — comes up often enough to flag by name: `deque(maxlen=3)` in Python keeps only the three most recently appended elements, silently discarding the oldest whenever a new one arrives past that limit, and it's the cleanest way to express a "keep the last K seen" requirement without manually checking a length and popping on every iteration. Chapter 6 leans on exactly this shape for windowed problems, and Chapter 11 builds an entirely different but equally important use of a deque or stack: maintaining the contents in a specific monotonic order (always increasing, or always decreasing) rather than simple insertion order, which is what makes the monotonic-stack pattern possible.

Hash maps and hash sets

A hash map stores key-value pairs and is built around a hash function: a function that converts a key into an integer, which is then used to decide which "bucket" of an underlying array the pair belongs in. When the hash function distributes keys well, looking up, inserting, or deleting a key is `O(1)` on average, because you compute the hash, jump straight to the relevant bucket, and — assuming that bucket holds only one or a small handful of entries — you're done. A hash set is the same idea holding only keys, no associated values, used purely to answer "have I seen this before?" in `O(1)` average time.

The average-case `O(1)` promise depends on handling collisions — cases where two different keys hash to the same bucket — gracefully, and it's worth knowing the two dominant strategies by name. Separate chaining keeps a small list (or similar structure) inside each bucket, and a colliding key is simply appended to that bucket's list; a lookup in a bucket then costs time proportional to that bucket's length, which is why a poor hash function that funnels many keys into the same bucket degrades performance toward `O(n)` in the worst case — every key ends up in one giant list. Open addressing instead keeps exactly one entry per underlying array slot, and

when a collision occurs, it probes forward through the array (using some deterministic sequence) until it finds an empty slot. Java's `HashMap` uses separate chaining (and, since Java 8, upgrades an individual bucket's list to a balanced tree once it grows past a threshold, which bounds the worst case for a single bucket at `O(log n)` instead of `O(n)`); Python's `dict` uses open addressing internally. You don't need to reimplement either strategy in an interview, but being able to explain, in one or two sentences, why a hash map degrades under bad hashing and how chaining or probing addresses it is a common, easy-to-lose follow-up question.

python

```
inventory_counts = {}
for item in ["bolt", "washer", "bolt", "nut", "bolt", "washer"]:
    inventory_counts[item] = inventory_counts.get(item, 0) + 1   # O(1) average

seen_serials = set()
duplicates = []
for serial in ["A1", "B2", "A1", "C3", "B2", "B2"]:
    if serial in seen_serials:          # O(1) average
        duplicates.append(serial)
    else:
        seen_serials.add(serial)
print(inventory_counts, duplicates)
```

java

```
Map<String, Integer> inventoryCounts = new HashMap<>();
for (String item : new String[]{"bolt", "washer", "bolt", "nut", "bolt", "washer"}) {
    inventoryCounts.merge(item, 1, Integer::sum);  // O(1) average
}

Set<String> seenSerials = new HashSet<>();
List<String> duplicates = new ArrayList<>();
for (String serial : new String[]{"A1", "B2", "A1", "C3", "B2", "B2"}) {
    if (!seenSerials.add(serial)) {  // O(1) average; add() returns false if present
        duplicates.add(serial);
    }
}
System.out.println(inventoryCounts + " " + duplicates);
```

One habit worth building now: Python's `dict.get(key, default)` and Java's `Map.merge` (or `getOrDefault`) both exist specifically to avoid writing an explicit "if the key isn't present yet, initialize it" branch every time you touch a counting map. Interview code that uses these idioms fluently reads noticeably cleaner than code that manually checks `if key not in counts` before every update, and clean, idiomatic counting-map code is one of the fastest ways to make a strong first impression in the opening minute of a round.

Two more pieces of vocabulary are worth having ready. A hash map's load factor is the ratio of stored entries to the number of underlying buckets; both Python and Java automatically trigger an internal resize — allocating a larger backing array and rehashing every existing entry into it — once the load factor crosses a threshold (Java's `HashMap` defaults to resizing at a load factor of `0.75`). That resize is the same amortized-cost story as the dynamic array from earlier in this chapter: individually expensive, rare enough in aggregate that insertion remains `O(1)` amortized. Second, be precise about what counts as a key: Java requires that any object used as a hash-map key correctly implement both `hashCode()` and

`equals()` in a mutually consistent way (two objects considered equal must produce the same hash code), and using a mutable object as a key — then mutating it after insertion — silently breaks lookups, because the object now hashes to a different bucket than the one it was actually stored in. Python has the analogous rule expressed differently: only hashable objects (roughly, immutable ones — strings, numbers, tuples of hashable elements) can be dictionary keys or set members at all, which is precisely why Python raises a `TypeError` if you try to use a `list` as a key — the language is refusing to let you create the Java bug by construction.

Binary trees and binary search trees

A binary tree is a node-based structure where each node holds a value and up to two child references, conventionally called left and right. A binary search tree (BST) adds one ordering invariant on top: for every node, every value in its left subtree is smaller, and every value in its right subtree is larger (assuming no duplicate values, or a consistent tie-breaking rule if duplicates are allowed). That invariant is what makes searching, inserting, and deleting all achievable in time proportional to the tree's height rather than its total size — at each node, comparing against the target tells you which subtree to descend into next, discarding the other subtree entirely.

The catch, and it's an important one for complexity statements: height is not the same as `log n` unless the tree is reasonably balanced. A BST built by inserting values in strictly increasing order degenerates into what is effectively a linked list, with height `O(n)` — every node has only a right child, and every operation becomes `O(n)`. A balanced BST, by contrast, has height `O(log n)`, and that's the case where search, insert, and delete are genuinely `O(log n)`. Self-balancing variants (red-black trees, AVL trees) exist specifically to guarantee that balance automatically; you won't be asked to implement one from scratch in a typical interview, but you should always state a BST's complexity as "`O(log n)` if balanced, `O(n)` in the worst case for a degenerate tree" rather than claiming `O(log n)` unconditionally.

python

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        self.root = self._insert(self.root, value)   # O(h), h = tree height

    def _insert(self, node, value):
        if node is None:
            return TreeNode(value)
        if value < node.value:
            node.left = self._insert(node.left, value)
        else:
            node.right = self._insert(node.right, value)
        return node
```

```
    def contains(self, value):
        node = self.root
        while node:                                    # O(h)
            if value == node.value:
                return True
            node = node.left if value < node.value else node.right
        return False

tree = BST()
for score in [50, 30, 70, 20, 40, 60, 80]:
    tree.insert(score)
print(tree.contains(40), tree.contains(45))
```

java

```
class TreeNode {
    int value;
    TreeNode left, right;
    TreeNode(int value) { this.value = value; }
}

class BST {
    TreeNode root;

    void insert(int value) {
        root = insert(root, value);      // O(h), h = tree height
    }

    private TreeNode insert(TreeNode node, int value) {
        if (node == null) return new TreeNode(value);
        if (value < node.value) node.left = insert(node.left, value);
        else node.right = insert(node.right, value);
        return node;
    }

    boolean contains(int value) {
        TreeNode node = root;
        while (node != null) {           // O(h)
            if (value == node.value) return true;
            node = value < node.value ? node.left : node.right;
        }
        return false;
    }
}
```

Chapter 12 devotes an entire chapter to traversal strategies (BFS with a queue, and the three flavors of DFS) and the recursive patterns built on top of trees; this section is deliberately just enough to make sure the vocabulary and the height-versus-size distinction are solid going in.

A related bit of vocabulary worth locking down now, because interviewers use it freely and expect you to follow along: a tree is balanced if the height difference between any node's left and right subtrees never exceeds some small bound (commonly one), guaranteeing `O(log n)` height overall; it's complete if every level is fully filled except possibly the last, which itself is filled left to right with no gaps (this is the shape a heap always maintains, which is exactly why a heap can be stored efficiently in a plain array rather than needing explicit child pointers); and it's full (sometimes called "proper" or "strict") if every node has either zero or two children, never exactly one. These three properties are independent of each other — a tree can be balanced without being complete, and complete without being full — and interviewers will sometimes hand you one of these terms as a constraint on the input ("assume the tree is a complete binary tree") specifically to see whether you

recognize what extra structure that buys you, so it's worth being able to distinguish all three instantly rather than treating "balanced" as a catch-all word for "well-behaved."

Heaps and priority queues

A heap is a tree-shaped structure — almost always implemented on top of a plain array rather than explicit node objects — that maintains one invariant: every parent is smaller than (or equal to) both of its children, for a min-heap, or larger than both, for a max-heap. That invariant doesn't fully sort the elements, but it guarantees that the smallest (or largest) element is always sitting at the root, retrievable in `O(1)`. Inserting a new element or removing the root both cost `O(log n)`, because both operations may need to walk up or down the tree's height to restore the invariant — and because a heap is kept balanced by construction (it's filled level by level, left to right), that height is always `O(log n)`, with none of the degeneration risk a BST has.

A priority queue is the abstract interface — "give me the highest-priority item first" — and a heap is simply the standard, efficient way to implement it. Python's `heapq` module provides a min-heap built directly on top of a plain list; Java's `PriorityQueue` is a min-heap by default and accepts a custom comparator when you need max-heap behavior or a non-trivial ordering rule.

python

```
import heapq

incident_queue = []
heapq.heappush(incident_queue, (2, "disk-latency"))    # O(log n)
heapq.heappush(incident_queue, (1, "service-down"))
heapq.heappush(incident_queue, (3, "cert-expiring"))

while incident_queue:
    priority, name = heapq.heappop(incident_queue)      # O(log n)
    print(priority, name)
```

java

```
record Incident(int priority, String name) {}

PriorityQueue<Incident> incidentQueue =
        new PriorityQueue<>((a, b) -> Integer.compare(a.priority(), b.priority()));
incidentQueue.offer(new Incident(2, "disk-latency"));   // O(log n)
incidentQueue.offer(new Incident(1, "service-down"));
incidentQueue.offer(new Incident(3, "cert-expiring"));

while (!incidentQueue.isEmpty()) {
    Incident next = incidentQueue.poll();                // O(log n)
    System.out.println(next.priority() + " " + next.name());
}
```

Both snippets print incidents back out in ascending priority order regardless of the order they were pushed in — `service-down` (priority 1) first, then `disk-latency` (priority 2), then `cert-expiring` (priority 3) — which is the entire point of the structure. Chapter 15 is built around recognizing when a problem is quietly asking for "the top K of something" or "always process the next-smallest item," both of which are heap problems in disguise.

Two implementation notes are worth knowing rather than rediscovering mid-interview. Python's `heapq` only provides min-heap behavior directly; the standard workaround for a max-heap is to negate values on the way in and out (push `-value`, and negate again when you pop), which looks like a minor trick the first time you see it but is genuinely the idiomatic approach, not a hack to apologize for. Second, a heap is emphatically not a fully sorted structure — it only guarantees that the root is the minimum (or maximum); the relative order of the other elements is unconstrained beyond the parent-smaller-than-children invariant, so don't assume you can read a heap's backing array left to right and get sorted output. If you need every element in sorted order, popping the root repeatedly (`O(n log n)` total) or using a dedicated sort is the correct move, not iterating the underlying array directly.

Graphs: adjacency list versus adjacency matrix

A graph is a set of nodes (often called vertices) connected by edges, and representing one in code comes down to a choice between two dominant strategies. An adjacency list stores, for each node, a list of the neighbors it directly connects to — implemented in practice as a hash map from node to a list (or set) of neighbors. An adjacency matrix instead stores a 2D grid of size `V × V`, where entry `[i][j]` records whether an edge exists between node `i` and node `j`.

The trade-off is memory versus lookup speed, and it maps directly onto how dense or sparse the graph is. An adjacency list uses `O(V + E)` memory — proportional to the number of nodes plus the number of edges actually present — which is efficient for sparse graphs (relatively few edges relative to the number of possible pairs), but checking whether a specific edge `(i, j)` exists requires scanning node `i`'s neighbor list, which costs `O(degree of i)` rather than `O(1)`. An adjacency matrix uses `O(V²)` memory regardless of how many edges actually exist, which wastes space on a sparse graph, but it answers "does this edge exist?" in `O(1)`, a flat array lookup. Most interview problems involving graphs — grids, network topologies, dependency chains — are sparse, which is why the adjacency list is the default representation throughout this book; reach for a matrix specifically when the graph is dense, or when repeated `O(1)` edge-existence checks matter more than memory.

python

```
from collections import defaultdict, deque

office_network = defaultdict(list)   # adjacency list: O(V + E) memory
edges = [("router-a", "router-b"), ("router-b", "router-c"), ("router-a", "router-c")]
for u, v in edges:
    office_network[u].append(v)
    office_network[v].append(u)

def bfs_order(graph, start):
    visited = {start}
    order = []
    queue = deque([start])
    while queue:                          # O(V + E) total across the traversal
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
```

```
                visited.add(neighbor)
                queue.append(neighbor)
    return order

print(bfs_order(office_network, "router-a"))
```

java

```
Map<String, List<String>> officeNetwork = new HashMap<>();   // adjacency list: O(V + E) memory
String[][] edges = {{"router-a", "router-b"}, {"router-b", "router-c"}, {"router-a", "router-c"}};
for (String[] e : edges) {
    officeNetwork.computeIfAbsent(e[0], k -> new ArrayList<>()).add(e[1]);
    officeNetwork.computeIfAbsent(e[1], k -> new ArrayList<>()).add(e[0]);
}

Set<String> visited = new HashSet<>(List.of("router-a"));
List<String> order = new ArrayList<>();
Deque<String> queue = new ArrayDeque<>(List.of("router-a"));
while (!queue.isEmpty()) {                // O(V + E) total across the traversal
    String node = queue.poll();
    order.add(node);
    for (String neighbor : officeNetwork.getOrDefault(node, List.of())) {
        if (visited.add(neighbor)) queue.add(neighbor);
    }
}
System.out.println(order);
```

Chapters 13 and 14 build the full traversal, topological-sort, union-find, and shortest-path toolkit on top of exactly this adjacency-list foundation; the goal here is just making sure the representation choice itself, and its $O(V + E)$ versus $O(V^2)$ trade-off, is second nature before those algorithms are layered on top.

A related decision worth flagging now, because it changes memory usage without changing which algorithms apply: directed versus undirected. The example above builds an undirected graph by adding each edge to both endpoints' neighbor lists, so a connection between `router-a` and `router-b` can be traversed in either direction. A directed graph — modeling something like task dependencies, where completing task A must precede task B but not the reverse — only adds the edge to the source node's list, and the choice of which one your problem calls for is exactly the kind of clarifying question worth asking before you start coding, since silently assuming the wrong direction produces a traversal that quietly visits the wrong set of nodes without throwing any error at all.

Tries: a brief preview

A trie (sometimes pronounced "try," to distinguish it from "tree") is a tree specialized for storing strings, where each edge represents a single character and each path from the root spells out a prefix. Two strings that share a common prefix share the same path through the trie for that prefix's length, which is what makes tries efficient for prefix-based operations — checking whether any stored word starts with a given prefix costs time proportional only to the length of the prefix itself, $O(L)$, completely independent of how many words are stored overall.

python

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False
```

```
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:                          # O(L), L = word length
            node = node.children.setdefault(ch, TrieNode())
        node.is_word = True

    def starts_with(self, prefix):
        node = self.root
        for ch in prefix:                         # O(L)
            if ch not in node.children:
                return False
            node = node.children[ch]
        return True

routes = Trie()
for path in ["/users", "/users/id", "/orders"]:
    routes.insert(path)
print(routes.starts_with("/users"), routes.starts_with("/billing"))
```

java

```
class TrieNode {
    Map<Character, TrieNode> children = new HashMap<>();
    boolean isWord = false;
}

class Trie {
    TrieNode root = new TrieNode();

    void insert(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {      // O(L), L = word length
            node = node.children.computeIfAbsent(ch, c -> new TrieNode());
        }
        node.isWord = true;
    }

    boolean startsWith(String prefix) {
        TrieNode node = root;
        for (char ch : prefix.toCharArray()) {     // O(L)
            node = node.children.get(ch);
            if (node == null) return false;
        }
        return true;
    }
}
```

This is deliberately just a preview — Chapter 21 covers tries in full depth, including using one to power an autocomplete system and combining one with backtracking for grid-based word search. What's worth taking away now is the core trade-off: a trie spends more memory than a plain hash set of strings (every shared prefix character gets its own node, and every node carries a map of possible next characters) in exchange for prefix queries that a hash set simply cannot answer efficiently at all — checking "does any stored word start with this prefix" against a hash set of whole words would require scanning every stored word, `O(total characters stored)`, versus a trie's `O(L)`.

The signal that should make you reach for a trie specifically, rather than a hash set, is the word "prefix" appearing anywhere in the problem statement — autocomplete, IP-address routing tables, dictionary validation during a word-search puzzle, anything where the question is about a family of strings that share a beginning rather

than about a single exact match. If the problem only ever asks "is this exact string present," a hash set answers that in `O(L)` (you still need to hash the whole string) with far less memory overhead, and reaching for a trie there would be over-engineering — a small but real judgment call worth naming out loud when you make it, since it demonstrates that you're choosing a structure for a reason rather than because it's the most recently practiced one.

Putting it side by side

The table below is meant as a reference you return to, not something to memorize in one sitting. "Average" reflects the typical case interviewers usually mean when they say "how fast is this"; "worst" reflects what can happen under adversarial input or an unbalanced structure, and stating the difference between the two, unprompted, is consistently a strong signal in an interview.

Structure	Access	Search	Insert	Delete	Notes
Array (static)	O(1)	O(n)	—	—	Fixed size, contiguous memory
Dynamic array	O(1)	O(n)	O(1) amortized (end) / O(n) (elsewhere)	O(n)	Doubling keeps append cheap on average
Singly linked list	O(n)	O(n)	O(1) at a known node	O(1) at a known node	No back-pointer; can't walk backward
Doubly linked list	O(n)	O(n)	O(1) at a known node	O(1) at a known node	Back-pointer enables O(1) removal without a predecessor search
Stack / queue / deque	O(n)	O(n)	O(1) at the relevant end	O(1) at the relevant end	Deque supports O(1) at both ends
Hash map / hash set	—	O(1) avg / O(n) worst	O(1) avg / O(n) worst	O(1) avg / O(n) worst	Worst case from poor hashing or heavy collisions
Balanced BST	O(log n)	O(log n)	O(log n)	O(log n)	Degrades to O(n) if unbalanced
Heap (binary)	O(1) for min/max	O(n) for arbitrary value	O(log n)	O(log n) for root	Not designed for arbitrary-element search
Graph (adjacency list)	—	O(degree) per node	O(1) per edge	O(degree) per edge	O(V + E) total memory
Graph (adjacency matrix)	O(1) per edge	O(1) per edge	O(1) per edge	O(1) per edge	$O(V^2)$ memory regardless of edge count
Trie	—	O(L)	O(L)	O(L)	L = length of the string; memory scales with shared-prefix structure

Every pattern chapter from here forward assumes you can look at a problem, recognize which one or two rows of this table are actually in play, and justify that

choice out loud in a sentence or two. That recognition step — not the coding that follows it — is usually what separates a candidate who solves a new problem confidently from one who stalls out searching for a starting point.

Chapter 4: A Repeatable Problem-Solving Framework

Knowing the patterns in Part II won't help you if you can't reliably get from "unfamiliar problem statement" to "pattern I recognize" under time pressure. That transition — the first ninety seconds to two minutes of a round, before you've written a single line of code — is where most interviews are actually won or lost. Candidates who ramble, guess, or start typing before they understand the problem tend to produce fragile solutions even when they technically know the right algorithm. Candidates who work through the same handful of steps every single time, regardless of the problem in front of them, tend to look calm, sound organized, and land on correct code even on problems they've never seen before.

This chapter gives you that repeatable sequence: a five-step framework called the C.L.E.A.R. method — Clarify, Log examples, Estimate approach, Analyze complexity, Refine and test. It isn't a magic trick that produces the optimal algorithm automatically. It's a discipline for spending your limited time in the right order, so that you never find yourself debugging code for a problem you didn't fully understand, and never find yourself staring silently at a blank editor with no plan at all. Every problem in the rest of this book can be approached through these five steps, and by the time you've worked through a few chapters of Part II, running through C.L.E.A.R. should feel less like a checklist and more like how you naturally think.

The five steps

Clarify. Before you write anything, restate the problem in your own words and ask about the details the prompt left ambiguous. What should happen on an empty input? Are there duplicate values, and if so, does that change the expected output? What's the realistic size of the input — hundreds of elements, or hundreds of millions? Can the input contain negative numbers, or values outside some expected range? This step feels slow the first few times you practice it deliberately, but it's rarely more than thirty to sixty seconds in a real interview, and it prevents the single most expensive mistake a candidate can make: solving the wrong problem confidently. It also does real communicative work — an interviewer watching you ask sharp, relevant questions is already updating their impression of you before you've written a line of code.

Log examples. Write down at least two concrete input/output pairs by hand, including at least one edge case — an empty input, a single-element input, or whatever boundary condition is most likely to break a careless solution. Do this before you commit to an approach, not after. Concrete examples turn a vague problem into something you can literally point at and reason about, and they double as your test suite once you have working code. If you find yourself unsure what the correct output for one of your own examples should be, that's valuable information: it usually

means the clarification step wasn't actually finished, and it's worth going back and asking one more question rather than guessing.

Estimate approach. Talk through more than one possible strategy before committing to one, even if the first idea that comes to mind is probably fine. State a brute-force approach explicitly, even if you don't plan to fully implement it — it establishes a correctness baseline and often reveals the shape of a faster approach once you see what the brute force is wasting effort on. Then name the pattern you suspect applies (a hash map for fast lookup, two pointers for a sorted array, a sliding window for a contiguous-subarray question) and say, out loud, why the signals in the problem point you there. This is the step where the pattern-recognition skills built throughout Part II pay off directly — the faster and more accurately you can name "this smells like a hash-based approach" or "this smells like two pointers," the less of your limited time gets spent groping for a starting point.

Analyze complexity. Before you write the full implementation, state the time and space complexity you expect the chosen approach to have, and briefly justify it — a single loop over the input suggests O(n); a nested loop or a sort-then-scan suggests something worse; an extra hash map suggests O(n) auxiliary space. Chapter 2 is entirely dedicated to making this step fast and accurate. Doing it here, before the code exists, matters for two reasons: it catches approaches that are obviously too slow before you've invested time writing them, and it gives the interviewer a checkpoint to weigh in — sometimes they'll tell you the naive complexity is fine for this problem, saving you the work of optimizing further; sometimes they'll nudge you that a faster approach exists, which is valuable information you want before you've written the slower version, not after.

Refine and test. Write the code, then verify it, deliberately, against the examples you logged in step two — trace through the logic by hand or mentally "run" it line by line, not just glance at the final output. Check the edge cases you identified during clarification. If you find a bug, don't panic and rewrite from scratch; isolate exactly which example fails, trace through that specific case, and fix the narrowest thing that explains the discrepancy. This step is also where you revisit the complexity estimate from step four and confirm the actual code matches what you predicted — if it doesn't, that gap is worth investigating and mentioning, because catching your own inefficiency unprompted is a strong, visible signal.

Five steps, and notice what they have in common: every one of them is something you do before fully trusting your own solution. That's deliberate. The framework isn't primarily about finding the cleverest algorithm — it's about making sure that whatever algorithm you land on is one you actually understood, checked, and can defend, rather than one you got lucky with.

Applying C.L.E.A.R. end to end

Here is a complete, original problem, worked through all five steps exactly as a candidate would experience them in a real round — including the internal monologue, not just the final answer.

The prompt, as given by the interviewer: "You're given a list of a small business's daily net cash-flow amounts — positive numbers for days with more income than expenses, negative numbers for days with more expenses than income. You're also given a target amount representing a reimbursement the business is trying to account for. Find the longest contiguous run of days whose total cash flow equals the target exactly, and return the start and end indices of that run. If no such run exists, indicate that clearly."

Clarify

The candidate's internal monologue: "Let me make sure I understand what 'contiguous' means here — I think it means consecutive days in the original order, not any subset of days I can pick freely. Let me also figure out what happens when nothing matches, and whether the amounts can be zero."

Out loud: "Just to confirm — by 'contiguous run of days' you mean a consecutive slice of the list, in the original day order, not an arbitrary subset of days I could pick? And can the daily amounts be zero, or only strictly positive or negative? What should I return if no contiguous run sums exactly to the target — is a sentinel value like `(-1, -1)` acceptable, or would you prefer something else?" The interviewer confirms: contiguous means consecutive in order, zero is a valid daily amount, and a `(-1, -1)` sentinel for "no match" is fine. The candidate also asks about input size — the interviewer says to assume up to a few hundred thousand days, which immediately rules out anything quadratic as a comfortable final answer.

Log examples

The candidate writes down, by hand, before deciding on an approach:

- `amounts = [40, 20, -15, 35, 10, -10, 20]`, `target = 45` → the run `[40, 20, -15]` (indices 0 through 2) sums to exactly 45, and no longer run does — expected output `(0, 2)`.
- `amounts = [5, 5, 5]`, `target = 100` → no contiguous run comes anywhere close to 100 — expected output `(-1, -1)`.
- `amounts = [-5, 8, -3, 8, -3, 12]`, `target = 12` → checking by hand, only the single day at index 5 (value `12`) sums to exactly 12 — expected output `(5, 5)`.

That last example is deliberately chosen to include negative numbers threaded through the middle of the array, since the candidate suspects negative values will be the detail most likely to break a careless approach.

Estimate approach

Internal monologue: "The obvious brute force is to check every possible contiguous run — for each starting day, walk forward summing until I either hit the target or the sum can't recover, tracking the longest match. That's checking roughly n² pairs of start and end points in the worst case, which the interviewer already told me won't fly at a few hundred thousand days. I need something that doesn't re-examine overlapping sums from scratch every time."

Out loud: "The brute-force approach is to check every start and end index pair directly, computing each subarray's sum from scratch — that's roughly O(n²), which is too slow for the input size you mentioned. I think there's a faster way using prefix sums. If I keep a running total of all amounts from the start of the array up to the current day, then the sum of any contiguous run from day `i+1` to day `j` is just `prefix[j] - prefix[i]`. I want that difference to equal the target, which means I'm looking for `prefix[i] = prefix[j] - target` for some earlier index `i`. If I store, in a hash map, the first index at which each prefix-sum value occurred, then at every day `j` I can check in O(1) whether the value `prefix[j] - target` has been seen before — and if it has, the run from right after that earlier index up through `j` sums exactly to the target. Tracking the longest such run as I scan once through the array gives me the answer in a single pass."

This is the moment the pattern gets named explicitly, out loud, which matters as much as the idea itself: a running-sum-plus-hash-map technique for subarray-sum problems, which recurs constantly enough across interview problems that recognizing it quickly is worth deliberate practice.

Analyze complexity

"This is a single pass over the array, doing O(1) work — a hash map lookup and a possible insert — at each step, so I expect O(n) time overall. Space-wise, the hash map can hold up to one entry per distinct prefix sum, so O(n) space in the worst case." The interviewer nods, confirming this is the complexity they were expecting, and the candidate moves to implementation with a validated plan rather than an assumption.

Refine and test

The candidate writes the following, narrating each piece as they go, and then traces all three logged examples through it by hand before declaring the solution finished.

Python solution:

python

```
def longest_reimbursable_window(daily_amounts, target):
    prefix_sum = 0
    first_seen = {0: -1}          # empty prefix (before day 0) sums to 0
    best_start, best_len = -1, 0

    for i, amount in enumerate(daily_amounts):
        prefix_sum += amount
```

```
        needed = prefix_sum - target
        if needed in first_seen:
            start = first_seen[needed] + 1
            length = i - start + 1
            if length > best_len:
                best_len = length
                best_start = start
        # only remember the FIRST time a prefix sum occurs, to keep runs as long as possible
        if prefix_sum not in first_seen:
            first_seen[prefix_sum] = i

    if best_start == -1:
        return (-1, -1)
    return (best_start, best_start + best_len - 1)
```

Java solution:

java

```
import java.util.HashMap;
import java.util.Map;

public class ReimbursableWindow {

    static int[] longestReimbursableWindow(int[] dailyAmounts, int target) {
        Map<Integer, Integer> firstSeen = new HashMap<>();
        firstSeen.put(0, -1);          // empty prefix (before day 0) sums to 0
        int prefixSum = 0;
        int bestStart = -1, bestLen = 0;

        for (int i = 0; i < dailyAmounts.length; i++) {
            prefixSum += dailyAmounts[i];
            int needed = prefixSum - target;
            if (firstSeen.containsKey(needed)) {
                int start = firstSeen.get(needed) + 1;
                int length = i - start + 1;
                if (length > bestLen) {
                    bestLen = length;
                    bestStart = start;
                }
            }
            // only remember the FIRST time a prefix sum occurs, to keep runs as long as possible
            firstSeen.putIfAbsent(prefixSum, i);
        }

        if (bestStart == -1) {
            return new int[]{-1, -1};
        }
        return new int[]{bestStart, bestStart + bestLen - 1};
    }
}
```

Tracing the first example by hand: prefix sums after each day are `40, 60, 45, 80, 90, 80, 100`. At day index 2 (prefix sum `45`), `needed = 45 - 45 = 0`, which is in `first_seen` at index `-1` — so the run spans indices `0` through `2`, length 3, matching the expected `(0, 2)`. Note also that at day index 5 (prefix sum `80`), `80` was already recorded at index 3, so it's not overwritten — this is exactly why the code only stores a prefix sum the first time it's seen: overwriting it with a later index would shrink any run that starts from that earlier occurrence, silently producing a shorter answer than the true longest one. Tracing the second example confirms no prefix-sum difference ever equals `100`, correctly returning `(-1, -1)`. Tracing the third example, with the interleaved negative values, confirms the single-day match at index `5` is found and no longer run is: this is precisely the case the candidate chose specifically to stress-test the negative-number handling, and watching it pass

confirms the fix generalizes rather than being a coincidence of the first example.

Complexity

Both solutions run in O(n) time — a single pass over the input, with O(1) work per element — and O(n) space in the worst case, since the hash map can hold up to one entry for every distinct prefix sum encountered, which is bounded by the number of days.

Common pitfalls

The most common mistake on this exact shape of problem is overwriting a prefix sum's stored index every time it recurs, rather than only the first time — doing so silently shortens the longest valid run without producing any error, which is exactly the kind of bug that only surfaces through deliberate tracing against a hand-built example, not through casual inspection of the code. A second common mistake is forgetting to seed the map with `{0: -1}` before the loop starts, which causes the algorithm to miss any valid run that begins at the very first day of the array, since there would be no earlier "empty prefix" recorded to compare against.

Interviewer follow-up

A natural extension: "What would you change if I asked for the number of contiguous runs summing to the target, rather than the single longest one?" The answer only requires a small adjustment — instead of storing the first index at which each prefix sum occurred, store a running count of how many times each prefix sum has occurred, and add that count to a running total every time a match is found, since every prior occurrence of the needed prefix sum marks the start of a separate valid run ending at the current day.

Why the framework, not just the answer, is the point

Notice that nothing about the C.L.E.A.R. method depended on already knowing the prefix-sum-and-hash-map technique in advance. The framework is what generated the chance to find that technique in the moment: clarifying ruled out ambiguous edge cases before they could cause a wrong answer, logging examples gave the candidate concrete cases to test against and to stress-test with negative numbers deliberately, estimating the approach forced an explicit brute-force baseline that made the faster idea's shape visible, analyzing complexity validated the plan before a single line of implementation was written, and refining and testing caught exactly the class of subtle bug — overwriting a prefix sum's index — that would otherwise have shipped silently broken.

This is the sequence to run, deliberately and completely, on every single problem in the chapters ahead, even the ones that feel immediately familiar. Skipping straight to code on a problem you do recognize is a common trap in its own right — it's how a

candidate confidently applies a memorized shape to a problem that only superficially resembles the one they're remembering, and misses a constraint that changes the right answer entirely. Run all five steps every time, and the pattern recognition you build through the rest of this book will have a reliable structure to plug into, rather than becoming just another thing to remember under pressure.

Part II: The Pattern Library

Nineteen chapters, one pattern each — the core of the book.